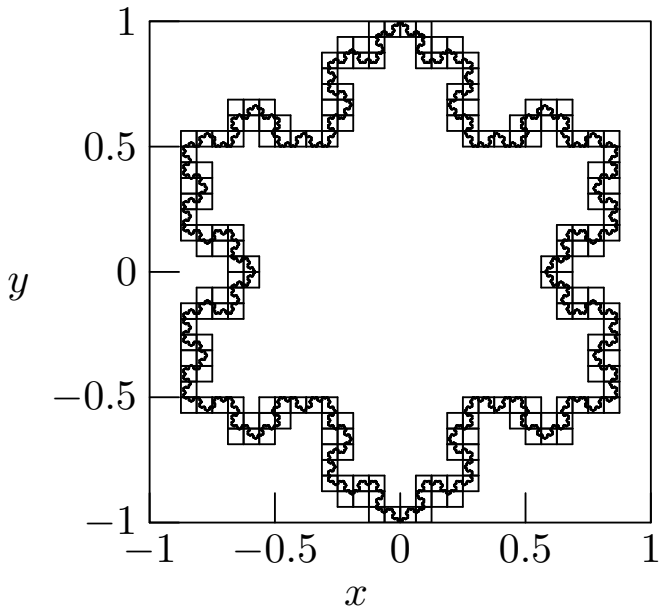
December 10, 2011 at 11:38

1. Introduction.

Boxcount

A program for calculating box-counting estimates to the fractal dimension of curves in the plane.
(Version 1.6 of November 26, 2011)

Written by Fredrik Jonsson



This CWEB[†] computer program calculates box-counting estimates of the fractal dimension of curves in the two-dimensional plane.

In the box-counting estimate to the fractal dimension of a graph in the domain $\{x, y : x_{\min} \le x \le x_{\max}, y_{\min} \le y \le y_{\max}\}$, a grid of squares, each of horizontal dimension $(x_{\max} - x_{\min})/2^m$ and vertical dimension $(y_{\max} - y_{\min})/2^m$, is superimposed onto the graph for integer numbers m. By counting the total number of such squares N_m needed to cover the entire graph at a given m (hence the term "box counting"), an estimate D_m to the fractal dimension D (or Hausdorff dimension) is obtained as $D_m = \ln(N_m)/\ln(2^m)$. This procedure may be repeated may times, with $D_m \to D$ as $m \to \infty$ for real fractal sets. However, for

[†] For information on the CWEB programming language by Donald E. Knuth, as well as samples of CWEB programs, see `http://www-cs-faculty.stanford.edu/~knuth/cweb.html`. For general information on literate programming, see `http://www.literateprogramming.com`.

finite-depth fractals (as generated by a computer), some limit on m is necessary in order to prevent trivial convergence towards $D_m \to 1$.

In addition to mere numerical calculation, the program also generates graphs of the box distributions, in form of METAPOST code which can be post-processed by other programs.

2. The CWEB programming language. For the reader who might not be familiar with the concept of the CWEB programming language, the following citations hopefully will be useful. For further information, as well as freeware compilers for compiling CWEB source code, see `http://www.literateprogramming.com`.

> *I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming.'*
>
> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> *The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.*

– Donald Knuth, *The CWEB System of Structured Documentation* (Addison-Wesley, Massachusetts, 1994)

> *The philosophy behind CWEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.*
>
> *The structure of a software program may be thought of as a 'WEB' that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TEX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages like C make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.*
>
> *Besides providing a documentation tool, CWEB enhances the C language by providing the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small sections and their local interrelationships. The CTANGLE program is so named because it takes a given web and moves the sections from their web structure into the order required by C; the advantage of programming in CWEB is that the algorithms can be expressed in "untangled" form, with each section explained separately. The CWEAVE program is so named because it takes a given web and intertwines the TEX and C portions contained in each section, then it knits the whole fabric into a structured document.*

– Donald Knuth, "Literate Programming", in *Literate Programming* (CSLI Lecture Notes, Stanford, 1992)

3. Revision history of the program.

2006-05-08 [v.1.0] <fj@phys.soton.ac.uk>
First properly working version of the BOXCOUNT program, written in CWEB and (ANSI-conformant) C. I have so far compiled the code with GCC using the `--pedantic` option and verified that the box-covering routine *get_num_covering_boxes_with_boxmaps*( ) and its more low-level core engine *box_intersection*( ) both work properly, by direct inspection of the compiled METAPOST graphs generated by this routine. That the numbers of counted boxes are correct has also been verified and the only remaining blocks to be added are related to the extraction of the fractal dimension as such. This first version of the BOXCOUNT program consists of 52671 bytes (1292 lines) of CWEB code, under CYGWIN generating an executable file of 22561 bytes and 33 pages of program documentation in 10 pt typeface.

2006-05-09 [v.1.1] <fj@phys.soton.ac.uk>
Changed the block for the estimate of the fractal dimension, so that the estimate now is obtained as the average of $\ln(N_{\text{boxes}})/\ln(2^n)$ for a set of n such that $N_{\min} \leq n \leq N_{\max}$, rather than performing a linear curve fit to the data. In order to sample statistical information on the estimate, such as standard deviation, average deviation and skewness, I incorporated a slightly modified version of the routine *moment*( ) from *Numerical Recipes in C*. Also added a preliminary section describing the test application of BOXCOUNT to the Koch fractal, being a simple test case which is easily implemented by means of a recursive generation of the data set for the input trajectory. As of today, the BOXCOUNT program consists of 59788 bytes (1444 lines) of CWEB code, under CYGWIN generating an executable file of 24253 bytes and 38 pages of program documentation in 10 pt typeface.

2006-05-14 [v.1.2] <fj@phys.soton.ac.uk>
Added a section on the command-line options for supplying the BOXCOUNT program with input parameters. Polished the section on the example of estimation of the box-counting dimension of the Koch fractal, and in particular changed the example from the Koch curve to the Koch snowflake instead, just for the sake of visual beauty. As of today, the BOXCOUNT program consists of 72560 bytes (1699 lines) of CWEB code, under OS X generating an executable file of 24996 bytes and 41 pages of program documentation in 10 pt typeface.

2006-05-17 [v.1.3] <fj@phys.soton.ac.uk>
Added documentation on the *get_num_covering_boxes_with_boxmaps*( ) routine and generally cleaned up in the documentation of the program. As of today, the BOXCOUNT program consists of 82251 bytes (1844 lines) of CWEB code, under CYGWIN generating an executable file of 29152 bytes and 40 pages of program documentation in 10 pt typeface.

2006-06-17 [v.1.4] <fj@phys.soton.ac.uk>
Added the `--graphsize` option, in order to override the default graph size. Also changed the inclusion of the input trajectory in the boxmaps, so that the BOXCOUNT program rather than using a METAPOST call of the form `gdraw "input.dat"` now chops the trajectory into proper pieces and includes the entire trajectory explicitly in the generated graph. This way the output METAPOST code naturally increases considerably in size, but is now at least self-sustaining even is separated from the original data file containing the input trajectory.

2006-10-25 [v.1.5] <fj@phys.soton.ac.uk>
Added two pages of text on the boxcounting estimate of the fractal dimension of the Koch snowflake fractal in the example section.

2011-11-26 [v.1.6] `<http://jonsson.eu>`
Updated `Makefile`:s for the generation of figures. Also corrected a rather stupid way of removing preceeding paths of file names.

4. Compiling the source code. The program is written in CWEB, generating ANSI C (ISO C90) conforming source code and documentation as plain TEX-source, and is to be compiled using the sequences as outlined in the `Makefile` listed below.

```
#
# Makefile designed for use with ctangle, cweave, gcc, and plain TeX.
#
# Copyright (C) 2002-2011, Fredrik Jonsson <http://jonsson.eu>
#
# The CTANGLE program converts a CWEB source document into a C program which
# may be compiled in the usual way.  The output file includes #line specifica-
# tions so that debugging can be done in terms of the CWEB source file.
#
# The CWEAVE program converts the same CWEB file into a TeX file that may be
# formatted and printed in the usual way.  It takes appropriate care of typo-
# graphic details like page layout and the use of indentation, italics,
# boldface, etc., and it supplies extensive cross-index information that it
# gathers automatically.
#
# CWEB allows you to prepare a single document containing all the information
# that is needed both to produce a compilable C program and to produce a well-
# formatted document describing the program in as much detail as the writer
# may desire.    The user of CWEB ought to be familiar with TeX as well as C.
#
PROJECT  = boxcount
CTANGLE  = ctangle
CWEAVE   = cweave
CC       = gcc
CCOPTS   = -O2 -Wall -ansi -std=iso9899:1990 -pedantic
LNOPTS   = -lm
TEX      = tex
DVIPS    = dvips
DVIPSOPT = -ta4 -D1200
METAPOST = mp
PS2PDF   = ps2pdf

all:  $(PROJECT) $(PROJECT).pdf
         @echo
"============================================================="
         @echo " To verify the execution performance of the BOXCOUNT program"
         @echo " on the Koch snowflake fractal, run 'make verification'."
         @echo
"============================================================="

$(PROJECT): $(PROJECT).o
         $(CC) $(CCOPTS) -o $(PROJECT) $(PROJECT).o $(LNOPTS)

$(PROJECT).o:  $(PROJECT).w
         $(CTANGLE) $(PROJECT)
         $(CC) $(CCOPTS) -c $(PROJECT).c
```

```
$(PROJECT).pdf:  $(PROJECT).ps
        $(PS2PDF) $(PROJECT).ps $(PROJECT).pdf

$(PROJECT).ps:  $(PROJECT).dvi
        $(DVIPS) $(DVIPSOPT) $(PROJECT).dvi -o $(PROJECT).ps

$(PROJECT).dvi:  $(PROJECT).w
        @make -C figures/
        @make -C kochxmpl/
        @make verification
        $(CWEAVE) $(PROJECT)
        $(TEX) $(PROJECT).tex

verification:
        @echo
"================================================================="
        @echo " Verifying the performance of the $(PROJECT) program on the
Koch"
        @echo " snowflake fractal of iteration order 11."
        @echo
"================================================================="
        make koch -C koch/
        make kochgraphs -C koch/
        make fractaldimension -C koch/

tidy:
        make -ik clean -C figures/
        make -ik clean -C koch/
        make -ik clean -C kochxmpl/
        -rm -Rf *  *.o *.exe *.dat *.tgz *.trj *.aux *.log *.idx *.scn *.dvi

clean:
        make -ik tidy
        -rm -Rf $(PROJECT) *.c *.pdf *mp *.toc *.tex *.ps

archive:
        make -ik tidy
        tar --gzip --directory=../ -cf ../$(PROJECT).tgz $(PROJECT)
```

This `Makefile` essentially executes two major calls. First, the CTANGLE program parses the CWEB source document `boxcount.w` to extract a C source file `boxcount.c` which may be compiled in the usual way using any ANSI C conformant compiler. The output source file includes `#line` specifications so that any debugging can be done conveniently in terms of the original CWEB source file. Second, the CWEAVE program parses the same CWEB source file to extract a plain TEX source file `boxcount.tex` which may be compiled in the usual way. It takes appropriate care of typographic details like page layout and the use of indentation, italics, boldface, and so on, and it supplies extensive cross-index information that it gathers automatically.

After having executed `make` in the same catalogue where the files `boxcount.w` and `Makefile` are located, one is left with an executable file `boxcount`, being the ready-to-use compiled program, and a PostScript file `boxcount.ps` which contains the full documentation of the program, that is to say the document you currently are reading. Notice that on platforms running Windows NT, Windows 2000, Windows ME, or any other operating system by Microsoft, the executable file will instead automatically be called `boxcount.exe`. This convention also applies to programs compiled under the UNIX-like environment CYGWIN.

5. Running the program. The program is entirely controlled by the command line options supplied when invoking the program. The syntax for executing the program is

```
boxcount [options]
```

where `options` include the following, given in their long as well as their short forms (with prefixes '`--`' and '`-`', respectively):

`--trajectoryfile`, `-i` ⟨*trajectory filename*⟩
: Specifies the input trajectory of the graph whose fractal dimension is to be estimated. The input file should describe the trajectory as two columns of x- and y-coordinates of the nodes, between which straight lines will interpolate the trajectory. Unless the boundary of the computational window is explicitly stated using the `-w` or `--computationwindow` options, the minimum and maximum x- and y-values of the specified trajectory will be used for the automatic internal computation of the proper computational domain boundaries.

`--outputfile`, `-o [append|overwrite]` ⟨*output filename*⟩
: Specifies the base name of the file to which the calculated output data will be written. If the `--outputfile` or `-o` option is followed by "`append`" the estimate for the fractal dimension will be appended to the file named ⟨*output filename*⟩.dat, which will be created if it does not exist. If the following word instead is "`overwrite`" the file will, of course, instead be overwritten.

`-w`, `--computationwindow llx` $\langle x_{\mathrm{LL}} \rangle$ `lly` $\langle y_{\mathrm{LL}} \rangle$ `urx` $\langle x_{\mathrm{UR}} \rangle$ `ury` $\langle y_{\mathrm{UR}} \rangle$
: This option explicitly specifies the domain over which the box-counting algorithm will be applied. By specifying this option, any automatic calculation of computational window will be neglected.

`--verbose`, `-v`
: Use this option to toggle verbose mode of the program execution, controlling the amount of information written to standard terminal output. (Default is off, that is to say quiet mode.)

`--boxmaps`, `-m`
: If this option is present, the BOXCOUNT program will generate METAPOST code for maps of the distribution of boxes, so-called "boxmaps". In doing so, also the input trajectory is included in the graphs. The convention for the naming of the output map files is that they are saved as ⟨*output filename*⟩.N.`dat`, where ⟨*output filename*⟩ is the base filename as specified using the `-o` or `--outputfile` option, N is the automatically appended current level of resolution refinement in the box-counting (that is to say, indicating the calculation performed for a $[2^N \times 2^N]$-grid of boxes), and where `dat` is a file suffix as automatically appended by the program. This option is useful for tracking the performance of the program, and for verification of the box counting algorithm.

`--graphsize` ⟨w⟩ ⟨h⟩
: If the `-m` or `--boxmaps` option is present at the command line, then the `--graphsize` option specifies the physical width w and height h in millimetres of the generated graphs, overriding the default sizes $w = 80\,\mathrm{mm}$ and $h = 80\,\mathrm{mm}$.

`--minlevel`, `-Nmin` $\langle N_{\mathrm{min}} \rangle$
: Specifies the minimum level N_{min} of grid refinement that will be used in the evaluation of the estimate of the fractal dimension. With this option specified, the coarsest level used in the box-counting will be a $[2^{N_{\mathrm{min}}} \times 2^{N_{\mathrm{min}}}]$-grid of boxes.

`--maxlevel`, `-Nmax` $\langle N_{\mathrm{max}} \rangle$
: This option is similar to the `--minlevel` or `-Nmin` options, with the difference that it instead specifies the maximum level N_{max} of grid refinement that will be used in the evaluation of the estimate of the fractal dimension. With this option specified, the finest level used in the box-counting will be a $2^{N_{\mathrm{max}}} \times 2^{N_{\mathrm{max}}}$-grid of boxes. If this option is omitted, a default value of $N_{\mathrm{max}} = 10$ will be used as default.

6. Application example: The Koch fractal. The Koch curve is one of the earliest described fractal curves, appearing in the 1904 article *Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire*, written by the Swedish mathematician Helge von Koch (1870–1924). In this application example, we employ the "snowflake" variant of the Koch fractal (merely for the sake of its beauty). The Koch snowflake fractal is constructed recursively using the following algorithm.

Algorithm A (*Construction of the Koch snowflake fractal*).

A1. [Create initiator.] Draw three line segments of equal length so that they form an equilateral triangle.

A2. [Line division.] Divide each of the line segments into three segments of equal length.

A3. [Replace mid segment by triangle.] Draw an equilateral triangle that has the middle segment from step one as its base.

A4. [Remove base of triangle.] Remove the line segment that is the base of the triangle from step A3.

A5. [Recursion step.] For each of the line segments remaining, repeat steps A2 through A4. ∎

The triangle resulting after step A1 is called the *initiator* of the fractal. After the first iteration of steps A1–A4, the result should be a shape similar to the Star of David; this is called the generator of the fractal. The Koch fractal resulting of the infinite iteration of the algorithm of construction has an infinite length, since each time the steps above are performed on each line segment of the figure there are four times as many line segments, the length of each being one-third the length of the segments in the previous stage. Hence, the total length of the perimeter of the fractal increases by 4/3 at each step, and for an initiator of total length L the total length of the perimeter at the nth step of iteration will be $(4/3)^n L$. The fractal dimension is hence $D = \ln 4/\ln 3 \approx 1.26$, being greater than the dimension of a line ($D = 1$) but less than, for example, Peano's space-filling curve ($D = 2$). The Koch fractal is an example of a curve which is continuous, but not differentiable anywhere. The area of the Koch snowflake is 8/5 that of the initial triangle, so an infinite perimeter encloses a finite area. The stepwise construction of the snowflake fractal is illustrated in Fig. 1.

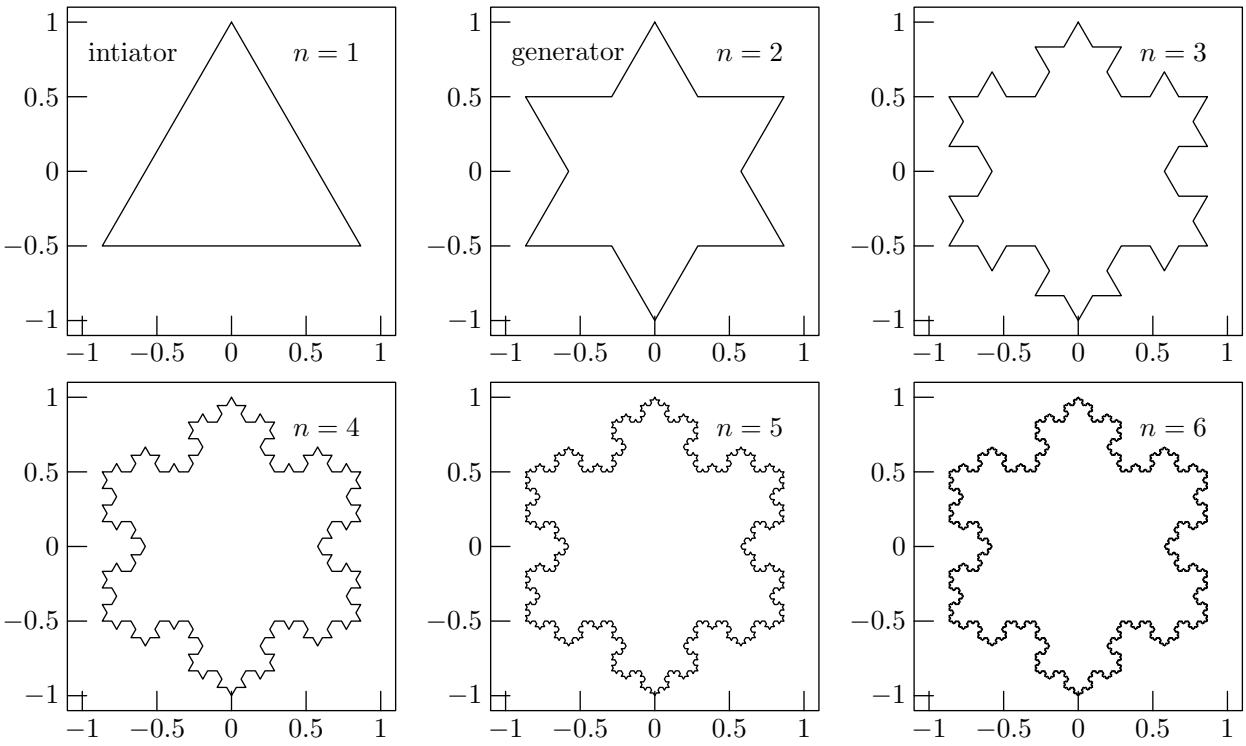


Figure 1. Construction of the Koch fractal of the "snowflake" type, in this case as inscribed in the unitary circle. For this case, the length of the initiator is $L = 3\sqrt{3}$ while its area is $A = L^2/(6\sqrt{3}) = 3\sqrt{3}/2$. For each fractal recursion depth n, the trajectory consists of a total of $3 \times 4^{(n-1)}$ connected linear segments.

The data set for the Koch fractal is straightforward to generate by means of recursion, as for example by using the following compact program (which in fact was used for the generation of the data sets for the Koch fractals on the previous page):

```
/*-----------------------------------------------------------------------
| File:  koch.c [ANSI-C conforming source code]
| Created:  May 8, 2006, Fredrik Jonsson <fj@phys.soton.ac.uk>
| Last modified:  May 8, 2006, Fredrik Jonsson <fj@phys.soton.ac.uk>
| Compile with:  gcc -O2 -g -Wall -pedantic -ansi koch.c -o koch -lm
| Description:  The KOCH program creates data sets corresponding to
|    the Koch fractal, for the purpose of acting as test objects for
|    the BOXCOUNT program.  The KOCH program is simply executed by
|    'koch <N>', where <N> is an integer describing the maximum
|    depth of recursion in the generation of the fractal data set.
|    If invoked without any arguments, <N>=6 is used as default.
|    The generated trajectory is written to standard output.
| Copyright (C) 2006 Fredrik Jonsson <fj@phys.soton.ac.uk>
=======================================================================*/
#include <math.h>
#include <stdio.h>
extern char *optarg;

void kochsegment(double xa,double ya,double xb,double yb,
   int depth,int maxdepth) {
   double xca,yca,xcb,ycb,xcc,ycc;
   if (depth==maxdepth) {
      fprintf(stdout,"%2.8f %2.8f\n",xb,yb);
   } else {
      xca=xa+(xb-xa)/3.0;
      yca=ya+(yb-ya)/3.0;
      xcb=xb-(xb-xa)/3.0;
      ycb=yb-(yb-ya)/3.0;
      xcc=(xa+xb)/2.0-(yb-ya)/(2.0*sqrt(3.0));
      ycc=(ya+yb)/2.0+(xb-xa)/(2.0*sqrt(3.0));
      kochsegment(xa,ya,xca,yca,depth+1,maxdepth);
      kochsegment(xca,yca,xcc,ycc,depth+1,maxdepth);
      kochsegment(xcc,ycc,xcb,ycb,depth+1,maxdepth);
      kochsegment(xcb,ycb,xb,yb,depth+1,maxdepth);
   }
}

int main(int argc, char *argv[]) {
   int maxdepth=6;
   if (argc>1) sscanf(argv[1],"%d",&maxdepth);
   if (maxdepth>0) {
      fprintf(stdout,"%2.8f %2.8f\n",0.0,1.0);
      kochsegment(0.0,1.0,sqrt(3.0)/2.0,-0.5,1,maxdepth);
      kochsegment(sqrt(3.0)/2.0,-0.5,-sqrt(3.0)/2.0,-0.5,1,maxdepth);
      kochsegment(-sqrt(3.0)/2.0,-0.5,0.0,1.0,1,maxdepth);
   }
   return(0);
}
```

The boxcounting dimension of the Koch snowflake fractal can now be investigated with assistance of the BOXCOUNT program. In the analysis as here presented, this is done using the following Makefile:

```
#
# Makefile designed for use with gcc, MetaPost and plain TeX.
#
# Copyright (C) 2002-2006, Fredrik Jonsson <fj@phys.soton.ac.uk>
#
CC       = gcc
CCOPTS   = -O2 -Wall -ansi -std=iso9899:1990 -pedantic
LNOPTS   = -lm
TEX      = tex
DVIPS    = dvips
METAPOST = mpost
#
# Define path and executable file for the BOXCOUNT program, used for
# calculating estimates of the box-counting fractal dimension of a
# trajectory in the plane.
#
BOXCOUNTPATH = ../
BOXCOUNT = $(BOXCOUNTPATH)/boxcount
#
# Define path and executable file for the KOCH program, used for generating
# the trajectory of the Koch snowflake fractal.
#
KOCHPATH = ../koch/
KOCH = $(KOCHPATH)/koch
all:  koch kochgen kochtab
koch:
        make -C ../koch/
kochgen:
        $(KOCH) 7 > koch.trj
        $(BOXCOUNT) --verbose --boxmaps --graphsize 42.0 42.0 \
                --computationwindow llx -1.1 lly -1.1 urx 1.1 ury 1.1 \
                --minlevel 3 --maxlevel 8 \
                --trajectoryfile koch.trj --outputfile overwrite koch
        for k in 03 04 05 06 07 08; do \
        $(METAPOST) koch.$$k.mp ;\
        $(TEX) -jobname=koch.$$k '\input epsf\nopagenumbers\
           \centerline\epsfxsize=120mm\epsfboxkoch.'$$k'.1\bye';\
        $(DVIPS) -D1200 -E koch.$$k -o koch.$$k.eps;\
        done

kochtab:
        $(KOCH) 7 > koch.trj
        $(BOXCOUNT) --verbose --minlevel 3 --maxlevel 14 \
                --computationwindow llx -1.1 lly -1.1 urx 1.1 ury 1.1 \
                --trajectoryfile koch.trj --outputfile overwrite koch

clean:
        -rm -Rf koch *  *.o *.exe *.dat *.mp *.mpx *.trj
        -rm -Rf *.tex *.aux *.log *.toc *.idx *.scn *.dvi *.1 *.eps
```

Having executed the `Makefile` as displayed in the previous page, where a recursion depth of $n = 7$ is used for the generation of the Koch fractal, we are left with a set of images of the consecutively refined grids in the boxcounting algorithm, and a table containing the estimates of the boxcounting dimension of the Koch snowflake fractal. In Fig. 2 the resulting maps of the boxes used in the boxcounting algorithm for refinement levels $m = 3, 4, \ldots, 8$ are shown, and as the grid refinement is finer and finer, the boxes finally will be barely visible in the limited resolution of computer graphics, on screen as well as in the generated Encapsulated PostScript code.

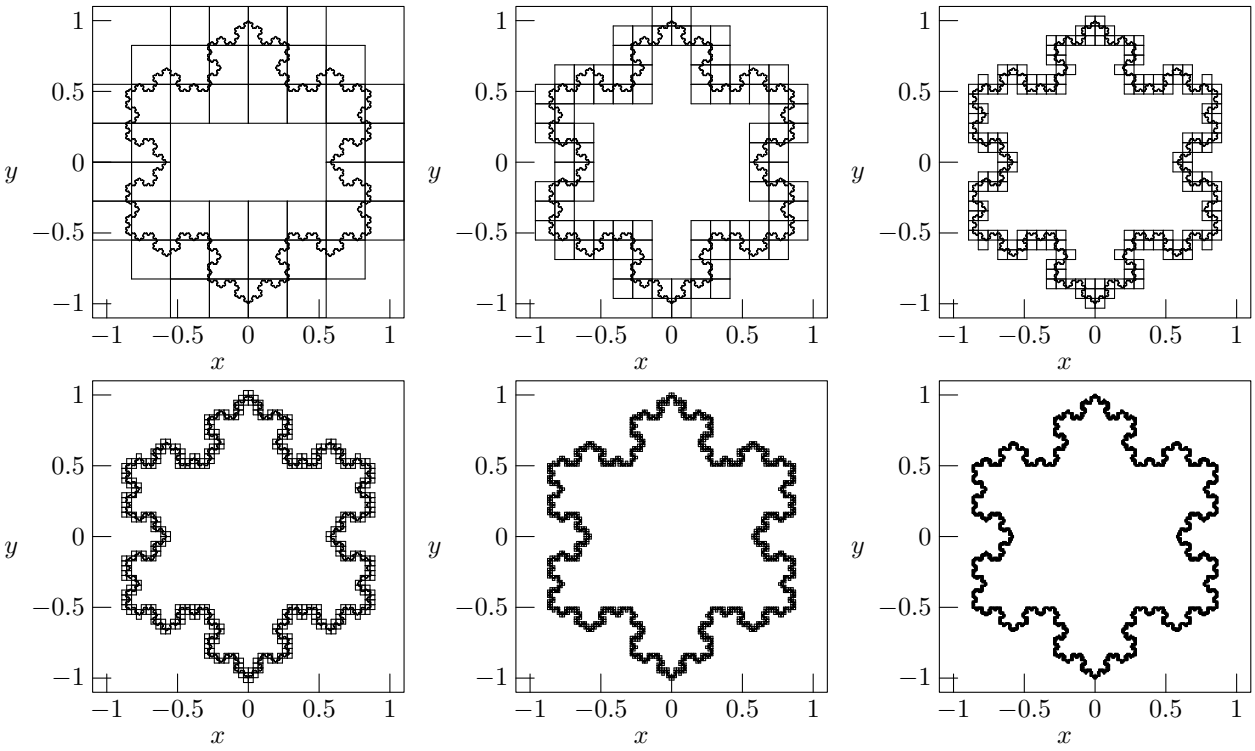


Figure 2. Consecutive steps of refinement $m = 3, 4, \ldots, 8$ of the grid of boxes covering the input trajectory, in this case a Koch snowflake fractal of recursion depth $n = 7$ (see Fig. 1). At each step m of refinement, a virtual grid of $2^m \times 2^m$ boxes is superimposed on the trajectory (fractal) and the number N_m of boxes covering the trajectory are counted. For the Koch snowflake fractal of recursion depth $n = 7$, the trajectory consists of a total of $3 \times 4^{(7-1)} = 12288$ connected linear segments.

The estimated boxcounting dimension $D_m = \ln(N_m)/\ln(2^m)$, with N_m as previously denoting the number of boxes in a $2^m \times 2^m$-grid required to cover the entire curve, is displayed in Table A.1. The values for the estimates could be compared with the analytically obtained value of $D = \ln 4/\ln 3 \approx 1.26$ for the fractal dimension. However, it should be emphasized that the box counting dimension here just is an estimate of one definition of the fractal dimension, which in many cases do not equal to other measures, and that we in the computation of the estimate always will arrive at a truncated result due to a limited precision and a limited amount of memory resources and computational time. As can be seen in the table, the initial estimates at lower resolutions are pretty crude, but in the final estimate we nevertheless end up with the box counting estimate of the fractal dimension as 1.29, which is reasonably close to the analytically obtained value of $D \approx 1.26$. Of course, further refinements such as Richardson extrapolation could be applied to increase the accuracy, but this is outside of the scope of the BOXCOUNT program, which only serves to provide the raw, basic algorithm of boxcounting.†

Table A.1 Boxcounting estimates of the fractal dimension of the Koch snowflake fractal of recursion order $n = 7$. In the table, m is the refinement order as indicated in the graphs in Fig. 2, N_m is the number of covering boxes counted at refinement level m, and $D_m = \ln(N_m)/\ln(2^m)$ is the estimate of the boxcounting dimension.

m	N_m	$D_m = \ln(N_m)/\ln(2^m)$
3	44	1.8198
4	96	1.6462
5	196	1.5229
6	504	1.4962
7	1180	1.4578
8	2856	1.4350
9	6844	1.4156
10	15620	1.3931
11	32320	1.3618
12	66200	1.3345
13	133600	1.3098
14	268804	1.2883

† For discussions on different definitions of the fractal dimension, see the English Wikipedia section on the Minkowski–Bouligand dimension, `http://en.wikipedia.org/wiki/Minkowski-Bouligand_dimension`.

7. The main program. Here follows the general outline of the main program.

```
⟨ Library inclusions 8 ⟩
⟨ Global definitions 9 ⟩
⟨ Global variables 10 ⟩
⟨ Subroutines 23 ⟩
int main(int argc, char *argv[])
{
  ⟨ Declaration of local variables 11 ⟩
  ⟨ Initialize variables 12 ⟩
  ⟨ Parse command line for parameters 13 ⟩
  ⟨ Display starting time of program execution 14 ⟩
  ⟨ Load input trajectory from file 15 ⟩
  ⟨ Open file for output of logarithmic estimate 16 ⟩
  ⟨ Extract boundary of global window of computation from input trajectory 17 ⟩
  ⟨ Get number of boxes covering the trajectory for all levels of refinement in resolution 18 ⟩
  ⟨ Compute the logarithmic estimate of the fractal dimension 19 ⟩
  ⟨ Save or append the logarithmic estimate to output file 20 ⟩
  ⟨ Close file for output of logarithmic estimate 21 ⟩
  ⟨ Display elapsed execution time 22 ⟩
  return (SUCCESS);
}
```

8. Library dependencies. The standard ANSI C libraries included in this program are:

`math.h`	For access to common mathematical functions.
`stdio.h`	For file access and any block involving *fprintf*.
`stdlib.h`	For access to the *exit* function.
`string.h`	For string manipulation, *strcpy*, *strcmp* etc.
`ctype.h`	For access to the *isalnum* function.

```
⟨ Library inclusions 8 ⟩ ≡
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

This code is used in section 7.

9. Global definitions. There are just a few global definitions present in the BOXCOUNT program:

`VERSION`	The current program revision number.
`COPYRIGHT`	The copyright banner.
`SUCCESS`	The return code for successful program termination.
`FAILURE`	The return code for unsuccessful program termination.
`NCHMAX`	The maximum number of characters allowed in strings for storing file names, including path.
`APPEND`	Code for the flag *output_mode*, used to determine if output should append to an existing file.
`OVERWRITE`	Code for the flag *output_mode*, used to determine if output should overwrite an existing file.

⟨ Global definitions 9 ⟩ ≡

```
#define VERSION "1.6"
#define COPYRIGHT "Copyright␣(C)␣2006-2011,␣Fredrik␣Jonsson"
#define SUCCESS (0)
#define FAILURE (1)
#define NCHMAX (256)
#define APPEND 1
#define OVERWRITE 2
```

This code is used in section 7.

10. Declaration of global variables. The only global variables allowed in my programs are *optarg*, which is a pointer to the the string of characters that specified the call from the command line, and *progname*, which simply is a pointer to the string containing the name of the program, as it was invoked from the command line.

⟨ Global variables 10 ⟩ ≡

```
extern char *optarg;
char *progname;
```

This code is used in section 7.

11. Declaration of local variables of the *main* program. In CWEB one has the option of adding variables along the program, for example by locally adding temporary variables related to a given sub-block of code. However, the philosophy in the BOXCOUNT program is to keep all variables of the *main* section collected together, so as to simplify tasks as, for example, tracking down a given variable type definition. The local variables of the program are as follows:

verbose	Boolean flag which, if being nonzero, tells the program to display information at terminal output during program execution.
user_set_compwin	Boolean flag which indicates whether the user has explicitly stated a window of computation or not.
output_mode	Variable which states whether the estimate of fractal dimension should be appended to an existing file which is created if it does not exist (for *output_mode* = `APPEND`), or if the data should overwrite already existing data in the file (for *output_mode* = `OVERWRITE`).
make_boxmaps	If nonzero, then graphs showing the distribution of covering boxes will be generated.
$*$*num_boxes*	Pointer to array holding the number of boxes at various depths of division.
initime	The time at which the BOXCOUNT program was initialized.
now	Dummy variable for extraction of current time from the system clock.
mm	The number of points M in the input trajectory. This is the length of the vectors *x_traj* and *y_traj* as described below.
nn	Gives the number of boxes in the x- or y-directions as 2^N.
nnmax	The maximum refinement depth $N_{\max}$ of N.
nnmin	The minimum refinement depth $N_{\min}$ of N.
global_llx, *global_lly*	Lower-left coordinates of global window of computation.
global_urx, *global_ury*	Upper-right coordinates of global window of computation.
$*$*x_traj*,$*$*y_traj*	Vectors containing the x- and y-values of the coordinates along the input trajectory.
$*x$,$*y$	Variables for keeping the refinement and number of boxes. (This needs to be changed as they are easily confused with x and y coordinates of the trajectory.)
$*$*trajectory_file*	Input file pointer, for reading the trajectory whose fractal dimension is to be estimated.
$*$*frac_estimate_file*	Output file pointer, for saving the estimated fractal dimension of the input trajectory.
$*$*boxmap_file*	Output file pointer, for saving maps of the spatial locations of the boxes covering the trajectory.
boxgraph_width	The physical width in millimetres of the generated METAPOST boxmap graphs.
boxgraph_height	The physical height in millimetres of the generated METAPOST boxmap graphs.
trajectory_filename	String for keeping the name of the file containing the input trajectory.
output_filename	String for keeping the base name of the set of output files.
frac_estimate_filename	String keeping the name of the file to which the estimate of the fractal dimension will be saved.
boxmap_filename	String for keeping the name of the file to which METAPOST code for maps of the spatial distribution of boxes will be written.
no_arg	Variable for extracting the number of input arguments present on the command line as the program is executed.
i	Dummy variable used in loops when reading the input trajectory.

$*fracdimen\_estimates$	Vector keeping the values characterizing estimated fractal dimension for various orders of N.
$ave, adev, sdev$	The average, average deviation, and standard deviation of the estimated fractal dimension for various orders of refinement N, as stored in *fracdimen_estimates*.
$var, skew, curt$	The variance, skewness, and kurtosis of the estimated fractal dimension for various orders of refinement N, as stored in *fracdimen_estimates*.

Generally in this program, the maximum number of characters a file name string can contain is `NCHMAX`, as defined in the definitions section of the program.

⟨ Declaration of local variables 11 ⟩ ≡

```
short verbose, user_set_compwin, output_mode, make_boxmaps;
long int *num_boxes;
time_t initime;
time_t now;
long mm, nn, nnmin, nnmax;
double global_llx, global_lly, global_urx, global_ury;
double *x_traj, *y_traj, *x, *y;
FILE *trajectory_file, *frac_estimate_file, *boxmap_file;
char trajectory_filename[NCHMAX], output_filename[NCHMAX], frac_estimate_filename[NCHMAX];
char boxmap_filename[NCHMAX];
double boxgraph_width, boxgraph_height;
int no_arg;
long i;
double *fracdimen_estimates, ave, adev, sdev, var, skew, curt;
```

This code is used in section 7.

12. Initialization of variables.

⟨ Initialize variables 12 ⟩ ≡

```
verbose = 0;      /* Verbose mode is off by default */
user_set_compwin = 0;      /* Before the command-line is parsed, nothing is known of these settings */
output_mode = OVERWRITE;      /* default mode is to overwrite existing files */
make_boxmaps = 0;      /* Default is to not create graphs of the box distributions */
nnmin = 0;      /* Default value for N_min */
nnmax = 10;      /* Default value for N_max */
strcpy(output_filename, "out");      /* Default output file basename. */
strcpy(trajectory_filename, "");      /* To indicate that no filename has been set yet. */
boxgraph_width = 80.0;      /* Default graph width in millimetres */
boxgraph_height = 56.0;      /* Default graph height in millimetres */
trajectory_file = Λ;
frac_estimate_file = Λ;
boxmap_file = Λ;
initime = time(Λ);      /* Time of initialization of the program. */
```

This code is used in section 7.

13. Parsing command line options. All input parameters are passed to the program through command line options and arguments to the program. The syntax of command line options is listed whenever the program is invoked without any options, or whenever any of the `--help` or `-h` options are specified at startup.

⟨ Parse command line for parameters 13 ⟩ ≡

```
{
  progname = strip_away_path(argv[0]);
  fprintf(stdout, "This is %s v.%s. %s\n", progname, VERSION, COPYRIGHT);
  no_arg = argc;
  while (--argc) {
    if (¬strcmp(argv[no_arg − argc], "-o") ∨ ¬strcmp(argv[no_arg − argc], "--outputfile")) {
      --argc;
      if (¬strcmp(argv[no_arg − argc], "append") ∨ ¬strcmp(argv[no_arg − argc], "a")) {
        output_mode = APPEND;
      }
      else if (¬strcmp(argv[no_arg − argc], "overwrite") ∨ ¬strcmp(argv[no_arg − argc], "o")) {
        output_mode = OVERWRITE;
      }
      else {
        fprintf(stderr, "%s: Error in '-o' or '--outputfile' option!", progname);
        exit(FAILURE);
      }
      --argc;
      strcpy(output_filename, argv[no_arg − argc]);
    }
    else if (¬strcmp(argv[no_arg − argc], "-w") ∨ ¬strcmp(argv[no_arg − argc], "--computationwindow"))
      {
      user_set_compwin = 1;
      --argc;
      if (¬strcmp(argv[no_arg − argc], "llx")) {
        --argc;
        if (¬sscanf(argv[no_arg − argc], "%lf", &global_llx)) {
          fprintf(stderr, "%s: Error in parsing lower-left x-value.\n", progname);
          exit(FAILURE);
        }
      }
      else {
        fprintf(stderr, "%s: Error in computation window option\n", progname);
        fprintf(stderr, "%s: Expecting 'llx'\n", progname);
        exit(FAILURE);
      }
      --argc;
      if (¬strcmp(argv[no_arg − argc], "lly")) {
        --argc;
        if (¬sscanf(argv[no_arg − argc], "%lf", &global_lly)) {
          fprintf(stderr, "%s: Error in parsing lower-left y-value.\n", progname);
          exit(FAILURE);
        }
      }
      else {
        fprintf(stderr, "%s: Error in computation window option\n", progname);
        fprintf(stderr, "%s: Expecting 'lly'\n", progname);
        exit(FAILURE);
```

```
    }
    −− argc;
    if (¬strcmp(argv[no_arg − argc], "urx")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%lf", &global_urx)) {
        fprintf(stderr, "%s: Error in parsing lower-left x-value.\n", progname);
        exit(FAILURE);
      }
    }
    else {
      fprintf(stderr, "%s: Error in computation window option\n", progname);
      fprintf(stderr, "%s: Expecting 'urx'\n", progname);
      exit(FAILURE);
    }
    −− argc;
    if (¬strcmp(argv[no_arg − argc], "ury")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%lf", &global_ury)) {
        fprintf(stderr, "%s: Error in parsing lower-left y-value.\n", progname);
        exit(FAILURE);
      }
    }
    else {
      fprintf(stderr, "%s: Error in computation window option\n", progname);
      fprintf(stderr, "%s: Expecting 'ury'\n", progname);
      exit(FAILURE);
    }
  }
  else if (¬strcmp(argv[no_arg − argc], "-i") ∨ ¬strcmp(argv[no_arg − argc], "--trajectoryfile")) {
    −− argc;
    strcpy(trajectory_filename, argv[no_arg − argc]);
  }
  else if (¬strcmp(argv[no_arg − argc], "-v") ∨ ¬strcmp(argv[no_arg − argc], "--verbose")) {
    verbose = (verbose ? 0 : 1);
  }
  else if (¬strcmp(argv[no_arg − argc], "-h") ∨ ¬strcmp(argv[no_arg − argc], "--help")) {
    showsomehelp( );
    exit(SUCCESS);
  }
  else if (¬strcmp(argv[no_arg − argc], "-m") ∨ ¬strcmp(argv[no_arg − argc], "--boxmaps")) {
    make_boxmaps = 1;
  }
  else if (¬strcmp(argv[no_arg − argc], "--graphsize")) {
    −− argc;
    if (¬sscanf(argv[no_arg − argc], "%lf", &boxgraph_width)) {
      fprintf(stderr, "%s: Error in width of '--graphsize' option.\n", progname);
      exit(FAILURE);
    }
    −− argc;
    if (¬sscanf(argv[no_arg − argc], "%lf", &boxgraph_height)) {
      fprintf(stderr, "%s: Error in height of '--graphsize' option.\n", progname);
      exit(FAILURE);
```

```
        }
      }
      else if (¬strcmp(argv[no_arg − argc], "--minlevel") ∨ ¬strcmp(argv[no_arg − argc], "-Nmin")) {
        −−argc;
        if (¬sscanf(argv[no_arg − argc], "%ld", &nnmin)) {
          fprintf(stderr, "%s: Error in '--minlevel' or '-Nmin' option.\n", progname);
          exit(FAILURE);
        }
      }
      else if (¬strcmp(argv[no_arg − argc], "--maxlevel") ∨ ¬strcmp(argv[no_arg − argc], "-Nmax")) {
        −−argc;
        if (¬sscanf(argv[no_arg − argc], "%ld", &nnmax)) {
          fprintf(stderr, "%s: Error in '--maxlevel' or '-Nmax' option.\n", progname);
          exit(FAILURE);
        }
      }
      else {
        fprintf(stderr, "%s: Specified option '%s' invalid!\n", progname, argv[no_arg − argc]);
        showsomehelp( );
        exit(FAILURE);
      }
    }
  }
```

This code is used in section 7.

14. Display starting time of program execution.

⟨ Display starting time of program execution 14 ⟩ ≡

```
  {
    fprintf(stdout, "%s: Program execution started %s", progname, ctime(&initime));
  }
```

This code is used in section 7.

15. Load input trajectory from file. This is the section where the trajectory to be analyzed is loaded into the memory, and where the number M of input coordinates present in the input data is determined by a single call to the subroutine *num_coordinate_pairs*( ).

```
⟨ Load input trajectory from file 15 ⟩ ≡
  {
    if (¬strcmp(trajectory_filename, "")) {
      fprintf(stderr, "%s: No input trajectory specified!\n", progname);
      fprintf(stderr, "%s: Please use the '--trajectoryfile' option.\n", progname);
      fprintf(stderr, "%s: Use '--help' option to display a help message.\n", progname);
      exit(FAILURE);
    }
    if ((trajectory_file = fopen(trajectory_filename, "r")) ≡ Λ) {
      fprintf(stderr, "%s: Could not open %s for loading trajectory!\n", progname,
          trajectory_filename);
      exit(FAILURE);
    }
    mm = num_coordinate_pairs(trajectory_file);
    x_traj = dvector(1, mm);
    y_traj = dvector(1, mm);
    for (i = 1; i ≤ mm; i++) {
      fscanf(trajectory_file, "%lf", &x_traj[i]);      /* scan x-coordinate */
      fscanf(trajectory_file, "%lf", &y_traj[i]);      /* scan y-coordinate */
    }
    fclose(trajectory_file);
    if (verbose) {
      fprintf(stdout, "%s: Loaded %ld coordinate pairs from file '%s'.\n", progname, mm,
          trajectory_filename);
    }
  }
```

This code is used in section 7.

16. Opening files for output by the program.

```
⟨ Open file for output of logarithmic estimate 16 ⟩ ≡
  {
    if (¬strcmp(output_filename, "")) {
      fprintf(stderr, "%s: No output base name specified!\n", progname);
      fprintf(stderr, "%s: Please use the '--outputfile' option.\n", progname);
      exit(FAILURE);
    }
    sprintf(frac_estimate_filename, "%s.dat", output_filename);
    if (output_mode ≡ APPEND) {
      if ((frac_estimate_file = fopen(frac_estimate_filename, "a")) ≡ Λ) {
        fprintf(stderr, "%s: Could not open %s for output!\n", progname, frac_estimate_filename);
        exit(FAILURE);
      }
      fseek(frac_estimate_file, 0L, SEEK_END);      /* set file pointer to the end of the file */
    }
    else if (output_mode ≡ OVERWRITE) {
      if ((frac_estimate_file = fopen(frac_estimate_filename, "w")) ≡ Λ) {
        fprintf(stderr, "%s: Could not open %s for loading trajectory!\n", progname,
            frac_estimate_filename);
        exit(FAILURE);
      }
      fseek(frac_estimate_file, 0L, SEEK_SET);      /* set file pointer to the beginning of the file */
    }
    else {
      fprintf(stderr, "%s: Error: Output mode (output_mode) undefined!", progname);
      exit(FAILURE);
    }
  }
```

This code is used in section 7.

17. Extract global window of computation. This block will only be executed if there was no explicit computational window stated at startup of the program (that is to say, with the `-w` or `--computationwindow` option). In order to determine the minimum area covering the entire input trajectory, this section scans sequentially through the trajectory and finds the minimum and maximum of the x- and y-coordinates. These values then form the lower-left and upper-right corner coordinates $(global\_llx, global\_lly)$ and $(global\_urx, global\_ury)$.

⟨ Extract boundary of global window of computation from input trajectory 17 ⟩ ≡

```
{
  if (¬user_set_compwin) {
    global_llx = x_traj[1];
    global_lly = y_traj[1];
    global_urx = global_llx;
    global_ury = global_lly;
    for (i = 1; i ≤ mm; i++) {
      if (x_traj[i] > global_urx) global_urx = x_traj[i];
      if (y_traj[i] > global_ury) global_ury = y_traj[i];
      if (x_traj[i] < global_llx) global_llx = x_traj[i];
      if (y_traj[i] < global_lly) global_lly = y_traj[i];
    }
    if (verbose) {
      fprintf(stdout, "%s: Global box-counting window:\n", progname);
      fprintf(stdout, "%s:    (llx,lly)=(%2.8f,%2.8f)\n", progname, global_llx, global_lly);
      fprintf(stdout, "%s:    (urx,ury)=(%2.8f,%2.8f)\n", progname, global_urx, global_ury);
    }
  }
}
```

This code is used in section 7.

18. Get the number of boxes covering the trajectory for all levels of refinement. This is the main loop where the program iterates over all the refinement levels and meanwhile gather how many boxes are needed to cover the trajectory as function of the refinement index N for $N_{\min} \le N \le N_{\max}$. In the loop, the program starts with a grid of $[2^{N_{\min}} \times 2^{N_{\min}}]$ and ends with a grid of $[2^{N_{\max}} \times 2^{N_{\max}}]$ boxes, at each step of refinement increasing the number of boxes by a factor of four. At each level of increasing resolution, the program relies on the routines *get_num_covering_boxes_with_boxmaps*( ) or *get_num_covering_boxes*( ) to perform the actual box-counting. The results of the box counting are stored in the vector *num_boxes*, which is used later on in the actual extraction of the estimate of fractal dimension.

⟨ Get number of boxes covering the trajectory for all levels of refinement in resolution 18 ⟩ ≡

```
{
  num_boxes = livector(1, nnmax);
  nn = 1;
  for (i = 1; i ≤ nnmin − 1; i++) nn = 2 ∗ nn;      /∗ This leaves nn as 2^(N_min−1) ∗/
  for (i = nnmin; i ≤ nnmax; i++) {      /∗ For all levels in refinement of grid resolution ∗/
    nn = 2 ∗ nn;
    if (make_boxmaps) {      /∗ do we wish to generate METAPOST graphs of the box distribution? ∗/
      sprintf(boxmap_filename, "%s-%02ld.mp", output_filename, i);
      num_boxes[i] = get_num_covering_boxes_with_boxmaps(x_traj, y_traj, mm, i, global_llx, global_lly,
          global_urx, global_ury, boxmap_filename, boxgraph_width, boxgraph_height, trajectory_filename);
    }
    else {      /∗ if not, just use the regular number-crunching routine ∗/
      num_boxes[i] = get_num_covering_boxes(x_traj, y_traj, mm, i, global_llx, global_lly, global_urx,
          global_ury);
    }
    if (verbose) {
      fprintf(stdout, "%s:␣N=%ld␣(%ldx%ld-grid␣of␣boxes):␣", progname, i, nn, nn);
      fprintf(stdout, "Trajectory␣covered␣by␣%ld␣boxes\n", num_boxes[i]);
    }
  }
}
```

This code is used in section 7.

19. Compute the logarithmic estimate of the fractal dimension. Having completed the task of actually calculating the number of boxes necessary to cover the trajectory, for varying box dimensions of width $(global\_urx - global\_llx)/2^n$ and height $(global\_ury - global\_lly)/2^n$, for $n = 1, 2, \ldots, N$, the only remaining arithmetics is to actually calculate the estimate for the fractal dimension. This is done by performing the fit of the linear function $y = ax + b$ to the data set obtained with $2\ln(n)$ as abscissa and $\ln(num\_boxes(n))$ as ordinata.

Also in this block, we analyze whether the option *make_boxmaps* is set as true (1) or not (0), determining whether a graph of the number of boxes as function of division depth should be written to file as well.

In the fitting of the linear function $y = ax+b$ to the data set (x_1, y_1), (x_2, y_2),...,(x_N, y_N), the parameters a and b can be explicitly obtained from the summation formulas

$$a = \frac{1}{D}\Big(N\sum_{k=1}^{N} x_k y_k - \sum_{k=1}^{N} x_k \sum_{k=1}^{N} y_k\Big), \qquad b = \frac{1}{N}\Big(\sum_{k=1}^{N} y_k - a\sum_{k=1}^{N} x_k\Big),$$

where

$$D \equiv N\sum_{k=1}^{N} x_k^2 - \Big(\sum_{k=1}^{N} x_k\Big)^2.$$

```
⟨ Compute the logarithmic estimate of the fractal dimension 19 ⟩ ≡
  {
    x = dvector(nnmin, nnmax);
    y = dvector(nnmin, nnmax);
    fracdimen_estimates = dvector(nnmin, nnmax);
    nn = 1;
    for (i = 1; i ≤ nnmax; i++) {
      nn = 2 * nn;
      if (nnmin ≤ i) x[i] = log((double) nn);
    }
    for (i = nnmin; i ≤ nnmax; i++) y[i] = log(num_boxes[i]);
    for (i = nnmin; i ≤ nnmax; i++) fracdimen_estimates[i] = y[i]/x[i];
    if (verbose) {
      for (i = nnmin; i ≤ nnmax; i++) {
        fprintf(stdout, "%s: N=%ld, fractal dimension estimate=%f\n", progname, i,
            fracdimen_estimates[i]);
      }
    }
    moment(fracdimen_estimates, nnmin, nnmax, &ave, &adev, &sdev, &var, &skew, &curt);
    if (verbose) {
      fprintf(stdout, "%s: Estimate of fractal dimension: %f\n", progname, ave);
      fprintf(stdout, "%s: Average deviation: %f\n", progname, adev);
      fprintf(stdout, "%s: Standard deviation: %f\n", progname, sdev);
      fprintf(stdout, "%s: Skewness: %f\n", progname, skew);
    }
    free_livector(num_boxes, 1, nnmax);      /* release the memory occupied by num_boxes */
    free_dvector(fracdimen_estimates, nnmin, nnmax);
    free_dvector(x, nnmin, nnmax);      /* release the memory occupied by x */
    free_dvector(y, nnmin, nnmax);      /* release the memory occupied by y */
  }
```

This code is used in section 7.

20. Save the fractal dimension to file.

⟨ Save or append the logarithmic estimate to output file 20 ⟩ ≡

```
{
  if (output_mode ≡ APPEND) {
    fseek(frac_estimate_file, 0L, SEEK_END);
  }
  else if (output_mode ≡ OVERWRITE) {
    fseek(frac_estimate_file, 0L, SEEK_SET);
  }
  fprintf(frac_estimate_file, "%f␣%f␣%f\n", ave, sdev, skew);
}
```

This code is used in section 7.

21. Close any open output files.

⟨ Close file for output of logarithmic estimate 21 ⟩ ≡

```
{
  fclose(frac_estimate_file);
}
```

This code is used in section 7.

22. Display elapsed execution time.

⟨ Display elapsed execution time 22 ⟩ ≡

```
{
  now = time(Λ);
  if (verbose)
    fprintf(stdout, "%s:␣Total␣execution␣time:␣%d␣s\n", progname, ((int) difftime(now, initime)));
  fprintf(stdout, "%s:␣Program␣execution␣closed␣%s", progname, ctime(&now));
}
```

This code is used in section 7.

23. Subroutines. In this section, all subroutines as used by the main program are listed.

⟨ Subroutines 23 ⟩ ≡
⟨ Routine for computation of average, average deviation and standard deviation 24 ⟩
⟨ Routine for obtaining the number of coordinate pairs in a file 25 ⟩
⟨ Routines for removing preceding path of filenames 26 ⟩
⟨ Routines for memory allocation of vectors and matrices 29 ⟩
⟨ Routine for displaying help message 38 ⟩
⟨ Routine for determining whether two lines intersect or not 39 ⟩
⟨ Routine for determining whether a line and a box intersect or not 40 ⟩
⟨ Routines for calculation of number of boxes covering the trajectory 41 ⟩

This code is used in section 7.

24. Routine for computation of average, average deviation and standard deviation. This routine is adopted from *Numerical Recipes in C*.

⟨ Routine for computation of average, average deviation and standard deviation 24 ⟩ ≡

```
void moment(double data[],int nnmin,int nnmax,double *ave,double *adev,double
        *sdev,double *var,double *skew,double *curt)
{
  int j;
  double ep = 0.0, s, p;
  if (nnmax - nnmin ≤ 1) {
    fprintf(stderr,"%s: Error in routine moment()! ",progname);
    fprintf(stderr,"(nnmax-nnmin>1 is a requirement)\n");
    exit(FAILURE);
  }
  s = 0.0;
  for (j = nnmin; j ≤ nnmax; j++) s += data[j];
  *ave = s/(nnmax - nnmin + 1);
  *adev = (*var) = (*skew) = (*curt) = 0.0;
  for (j = nnmin; j ≤ nnmax; j++) {
    *adev += fabs(s = data[j] - (*ave));
    ep += s;
    ep += s;
    *var += (p = s * s);
    *skew += (p *= s);
    *curt += (p *= s);
  }
  *adev /= (nnmax - nnmin + 1);
  *var = (*var - ep * ep/(nnmax - nnmin + 1))/(nnmax - nnmin);
  *sdev = sqrt(*var);
  if (*var) {
    *skew /= ((nnmax - nnmin + 1) * (*var) * (*sdev));
    *curt = (*curt)/((nnmax - nnmin + 1) * (*var) * (*var)) - 3.0;
  }
  else {
    fprintf(stderr,"%s: Error in routine moment()! ",progname);
    fprintf(stderr,"No skew/kurtosis for zero variance\n");
    exit(FAILURE);
  }
}
```

This code is used in section 23.

25. Routine for obtaining the number of coordinate pairs in a file. This routine is called prior to loading the trajectory, in order to get the size needed for allocating the memory for the trajectory vector. As the number of coordinates M has been established, two vectors *x_traj*$[1..M]$ and *y_traj*$[1..M]$, containing the coordinates (x_m, y_m) of the trajectory.

```
⟨ Routine for obtaining the number of coordinate pairs in a file 25 ⟩ ≡
  long int num_coordinate_pairs(FILE *trajectory_file)
  {
    double tmp;
    int tmpch;
    long int mm = 0;
    fseek(trajectory_file, 0L, SEEK_SET);      /* rewind file to beginning */
    while ((tmpch = getc(trajectory_file)) ≠ EOF) {
      ungetc(tmpch, trajectory_file);
      fscanf(trajectory_file, "%lf", &tmp);      /* Read away the x coordinate */
      fscanf(trajectory_file, "%lf", &tmp);      /* Read away the y coordinate */
      mm++;
      tmpch = getc(trajectory_file);      /* Read away blanks and linefeeds */
      while ((tmpch ≠ EOF) ∧ (¬isdigit(tmpch))) tmpch = getc(trajectory_file);
      if (tmpch ≠ EOF) ungetc(tmpch, trajectory_file);
    }
    fseek(trajectory_file, 0L, SEEK_SET);      /* rewind file to beginning */
    return (mm);
  }
```

This code is used in section 23.

26. Routines for removing preceding path of filenames. In this block all routines related to removing preceding path strings go. Not really fancy programming, and no contribution to any increase of numerical efficiency or precision; just for the sake of keeping a tidy terminal output of the program. The *strip_away_path*( ) routine is typically called when initializing the program name string *progname* from the command line string *argv*[0], and is typically located in the blocks related to parsing of the command line options.

```
⟨ Routines for removing preceding path of filenames 26 ⟩ ≡
  ⟨ Routine for checking valid path characters 27 ⟩
  ⟨ Routine for stripping away path string 28 ⟩
```

This code is used in section 23.

27. Checking for a valid path character. The *pathcharacter* routine takes one character *ch* as argument, and returns 1 ("true") if the character is valid character of a path string, otherwise 0 ("false") is returned.

```
⟨ Routine for checking valid path characters 27 ⟩ ≡
  short pathcharacter(int ch)
  {
    return (isalnum(ch) ∨ (ch ≡ '.') ∨ (ch ≡ '/') ∨ (ch ≡ '\\') ∨ (ch ≡ '_') ∨ (ch ≡ '-') ∨ (ch ≡ '+'));
  }
```

This code is used in section 26.

28. Routine for stripping away path string of a file name. The *strip_away_path*( ) routine takes a character string *filename* as argument, and returns a pointer to the same string but without any preceding path segments. This routine is, for example, useful for removing paths from program names as parsed from the command line.

```
⟨ Routine for stripping away path string 28 ⟩ ≡
  char *strip_away_path(char filename[])
  {
    int j, k = 0;
    while (pathcharacter(filename[k])) k++;
    j = (--k);      /* this is the uppermost index of the full path+file string */
    while (isalnum((int)(filename[j]))) j--;
    j++;      /* this is the lowermost index of the stripped file name */
    return (&filename[j]);
  }
```

This code is used in section 26.

29. Subroutines for memory allocation. Here follows the routines for memory allocation and deallocation of double precision real and complex valued vectors, as used for storing the optical field distribution in the grating, the refractive index distribution of the grating, etc.

```
⟨ Routines for memory allocation of vectors and matrices 29 ⟩ ≡
  ⟨ Routine for allocation of vectors of double precision 30 ⟩
  ⟨ Routine for deallocation of vectors of integer precision 33 ⟩
  ⟨ Routine for allocation of vectors of integer precision 32 ⟩
  ⟨ Routine for deallocation of vectors of double precision 31 ⟩
  ⟨ Routine for allocation of matrices of short integer precision 36 ⟩
  ⟨ Routine for deallocation of matrices of short integer precision 37 ⟩
```

This code is used in section 23.

30. The *dvector* routine allocates a real-valued vector of double precision, with vector index ranging from nl to nh.

```
⟨ Routine for allocation of vectors of double precision 30 ⟩ ≡
  double *dvector(long nl, long nh)
  {
    double *v;
    v = (double *) malloc((size_t)((nh - nl + 2) * sizeof(double)));
    if (¬v) {
      fprintf(stderr, "Error: Allocation failure in dvector()\n");
      exit(FAILURE);
    }
    return v - nl + 1;
  }
```

This code is used in section 29.

31. The *free_dvector* routine release the memory occupied by the real-valued vector $v[nl \,..\, nh]$.

```
⟨ Routine for deallocation of vectors of double precision 31 ⟩ ≡
  void free_dvector(double *v, long nl, long nh)
  {
    free((char *)(v + nl - 1));
  }
```

This code is used in section 29.

32. The *ivector* routine allocates a real-valued vector of integer precision, with vector index ranging from nl to nh.

```
⟨ Routine for allocation of vectors of integer precision 32 ⟩ ≡
  int *ivector(long nl, long nh)
  {
    int *v;
    v = (int *) malloc((size_t)((nh - nl + 2) * sizeof(int)));
    if (¬v) {
      fprintf(stderr, "Error: Allocation failure in ivector()\n");
      exit(FAILURE);
    }
    return v - nl + 1;
  }
```

See also section 34.

This code is used in section 29.

33. The *free_ivector* routine release the memory occupied by the real-valued vector $v[nl \mathinner{..} nh]$.

```
⟨ Routine for deallocation of vectors of integer precision 33 ⟩ ≡
  void free_ivector(int *v, long nl, long nh)
  {
    free((char *)(v + nl - 1));
  }
```

See also section 35.

This code is used in section 29.

34. The *livector* routine allocates a real-valued vector of long integer precision, with vector index ranging from nl to nh.

```
⟨ Routine for allocation of vectors of integer precision 32 ⟩ +≡
  long int *livector(long nl, long nh)
  {
    long int *v;
    v = (long int *) malloc((size_t)((nh - nl + 2) * sizeof(long int)));
    if (¬v) {
      fprintf(stderr, "Error: Allocation failure in livector()\n");
      exit(FAILURE);
    }
    return v - nl + 1;
  }
```

35. The *free_livector* routine release the memory occupied by the real-valued vector $v[nl \mathinner{..} nh]$.

```
⟨ Routine for deallocation of vectors of integer precision 33 ⟩ +≡
  void free_livector(long int *v, long nl, long nh)
  {
    free((char *)(v + nl - 1));
  }
```

36. The *simatrix* routine allocates an array of short integer precision, with array row index ranging from *nrl* to *nrh* and column index ranging from *ncl* to *nch*.

```
⟨ Routine for allocation of matrices of short integer precision 36 ⟩ ≡
  short int **simatrix(long nrl, long nrh, long ncl, long nch)
  {
    long i, nrow = nrh − nrl + 1, ncol = nch − ncl + 1;
    short int **m;
    m = (short int **) malloc((size_t)((nrow + 1) * sizeof(short int *)));
    if (¬m) {
      fprintf(stderr, "%s: Allocation failure 1 in simatrix()\n", progname);
      exit(FAILURE);
    }
    m += 1;
    m −= nrl;
    m[nrl] = (short int *) malloc((size_t)((nrow * ncol + 1) * sizeof(short int)));
    if (¬m[nrl]) {
      fprintf(stderr, "%s: Allocation failure 2 in simatrix()\n", progname);
      exit(FAILURE);
    }
    m[nrl] += 1;
    m[nrl] −= ncl;
    for (i = nrl + 1; i ≤ nrh; i++) m[i] = m[i − 1] + ncol;
    return m;
  }
```

This code is used in section 29.

37. The *free_simatrix* routine releases the memory occupied by the short integer matrix $v[nl \,..\, nh]$, as allocated by *simatrix*( ).

```
⟨ Routine for deallocation of matrices of short integer precision 37 ⟩ ≡
  void free_simatrix(short int **m, long nrl, long nrh, long ncl, long nch)
  {
    free((char *)(m[nrl] + ncl − 1));
    free((char *)(m + nrl − 1));
  }
```

This code is used in section 29.

38. Routine for displaying help message to standard terminal output.

```
⟨ Routine for displaying help message 38 ⟩ ≡
  void showsomehelp(void)
  {
    fprintf(stderr, "Usage: %s [options]\n", progname);
    fprintf(stderr, "Options:\n");
    fprintf(stderr, " -h, --help\n");
    fprintf(stderr, "     Display this help message and exit clean.\n");
    fprintf(stderr, " -v, --verbose\n");
    fprintf(stderr, "     Toggle verbose mode on/off.\n");
    fprintf(stderr, " -o, --outputfile <str>\n");
    fprintf(stderr, "     Specifies the base name of the output files where the program\n");
    fprintf(stderr, "     is to save the calculated data. If the --outputfile or -o\n");
    fprintf(stderr, "     option is followed by 'append' the estimate for the fractal\n");
    fprintf(stderr, "     dimension will be appended to the file named <str>.dat, which\n");
    fprintf(stderr, "     will be created if it does not exist. If the following word\n");
    fprintf(stderr, "     instead is `overwrite' the file will instead be overwritten.\n");
    fprintf(stderr, " -i, --trajectoryfile\n");
    fprintf(stderr, "     Specifies the input trajectory of the graph whose fractal\n");
    fprintf(stderr, "     dimension is to be estimated.\n");
    fprintf(stderr, " -w, --computationwindow llx <num> lly <num> urx <num> ury <num>\n");
    fprintf(stderr, "     This option explicitly specifies the domain over which the\n");
    fprintf(stderr, "     box-counting algorithm will be applied, in terms of the\n");
    fprintf(stderr, "     lower-left and upper-right corners (llx,lly) and (urx,ury),\n");
    fprintf(stderr, "     respectively. By specifying this option, any automatic\n");
    fprintf(stderr, "     calculation of computational window will be neglected.\n");
    fprintf(stderr, " -m, --boxmaps\n");
    fprintf(stderr, "     If this option is present, the program will generate\n");
    fprintf(stderr, "     MetaPost code for maps of the distribution of boxes.\n");
    fprintf(stderr, "     In doing so, also the input trajectory is included in\n");
    fprintf(stderr, "     the graphs. The convention for the naming of the output\n");
    fprintf(stderr, "     map files is that they are saved as <str>.<N>.dat,\n");
    fprintf(stderr, "     where <str> is the base filename as specified using the\n");
    fprintf(stderr, "     -o or --outputfile option, <N> is the automatically appended\n");
    fprintf(stderr, "     current level of resolution refinement in the box-counting,\n");
    fprintf(stderr, "     and where '.dat' is the file suffix as automatically appended\n");
    fprintf(stderr, "     by the program.\n");
    fprintf(stderr, " --graphsize <width in mm> <height in mm>\n");
    fprintf(stderr, "     If the -m or --boxmaps option is present at the command line,\n");
    fprintf(stderr, "     then the --graphsize option will override the default graph\n");
    fprintf(stderr, "     size of the generated boxmaps. (Default graph size is 80 mm\n");
    fprintf(stderr, "     width and 56 mm height.)\n");
    fprintf(stderr, " -Nmin, --minlevel <num>\n");
    fprintf(stderr, "     Specifies the minimum level Nmin of grid refinement that \n");
    fprintf(stderr, "     will be used in the evaluation of the estimate of the fractal\n");
    fprintf(stderr, "     dimension. With this option specified, the coarsest level\n");
    fprintf(stderr, "     used in the box-counting will be a [(2^Nmin)x(2^Nmin)]-grid\n");
    fprintf(stderr, "     of boxes.\n");
    fprintf(stderr, " -Nmax, --maxlevel <num>\n");
    fprintf(stderr, "     This option is similar to the --minlevel or -Nmin options,\n");
    fprintf(stderr, "     with the difference that it instead specifies the maximum\n");
    fprintf(stderr, "     level Nmax of grid refinement that will be used in the\n");
```

```
    fprintf(stderr, "␣␣␣␣evaluation␣of␣the␣estimate␣of␣the␣fractal␣dimension.\n");
  }
```

This code is used in section 23.

39. The *lines_intersect*$(p1x, p1y, q1x, q1y, p2x, p2y, q2x, q2y)$ routine takes the start and end points of two line segments, the first between $(p1x, p1y)$ and $(q1x, q1y)$ and the second between $(p2x, p2y)$ and $(q2x, q2y)$, and returns 1 ('true') if they are found to intersect, and 0 ('false') otherwise.

For a brief sumnmary of the algorithm behind this routine, consider two line segments in the plane, the first one between the points $\mathbf{p}_1$ and $\mathbf{q}_1$ and the second one between $\mathbf{p}_2$ and $\mathbf{q}_1$. In general, these segments can be written in parametric forms as $\mathbf{r}_1 = \mathbf{p}_1 + t_1(\mathbf{q}_1 - \mathbf{p}_1)$ and $\mathbf{r}_2 = \mathbf{p}_2 + t_2(\mathbf{q}_2 - \mathbf{p}_2)$ for $0 \le t_1 \le 1$ and $0 \le t_2 \le 1$. These line segments intersect each other if they for these intervals for the parameters t_1 and t_2 share a common point, or equivalently if the solutions to

$$\mathbf{p}_1 + t_1(\mathbf{q}_1 - \mathbf{p}_1) = \mathbf{p}_2 + t_2(\mathbf{q}_2 - \mathbf{p}_2) \qquad \Leftrightarrow \qquad (\mathbf{q}_1 - \mathbf{p}_1)t_1 + (\mathbf{p}_2 - \mathbf{q}_2)t_2 = \mathbf{p}_2 - \mathbf{p}_1$$

both simultaneously satisfy $0 \le t_1 \le 1$ and $0 \le t_2 \le 1$. This vectorial equation can equivalently be written in component form as

$$(q_{1x} - p_{1x})t_1 + (p_{2x} - q_{2x})t_2 = p_{2x} - p_{1x},$$
$$(q_{1y} - p_{1y})t_1 + (p_{2y} - q_{2y})t_2 = p_{2y} - p_{1y},$$

which after some straightforward algebra gives the explicit solutions for the parameters t_1 and t_2 as

$$t_1 = \frac{ed - bf}{ad - bc}, \qquad t_2 = \frac{af - ec}{ad - bc},$$

where

$$a \equiv (q_{1x} - p_{1x}), \qquad b \equiv (p_{2x} - q_{2x}), \qquad c \equiv (q_{1y} - p_{1y}),$$
$$d \equiv (p_{2y} - q_{2y}), \qquad e \equiv (p_{2x} - p_{1x}), \qquad f \equiv (p_{2x} - p_{1x}).$$

Hence, the two line segments intersect if and only if

$$0 \le \frac{ed - bf}{ad - bc} \le 1, \qquad \text{and} \qquad 0 \le \frac{af - ec}{ad - bc} \le 1.$$

By observing that their denominators are equal, the evaluation of these quotes involves in total 6 floating-point multiplications, 2 divisions and 3 subtractions. Notice that whenever $ad - bd = 0$, the two lines are parallell and will never intersect, regardless of the values of t_1 and t_2.

⟨ Routine for determining whether two lines intersect or not 39 ⟩ ≡

```
short int lines_intersect(double p1x, double p1y, double q1x, double q1y, double p2x, double
        p2y, double q2x, double q2y)
{
  double a, b, c, d, e, f, det, tmp1, tmp2;
  short int intersect;
  a = q1x − p1x;
  b = p2x − q2x;
  c = q1y − p1y;
  d = p2y − q2y;
  e = p2x − p1x;
  f = p2y − p1y;
  det = a ∗ d − b ∗ c;
  tmp1 = e ∗ d − b ∗ f;
  tmp2 = a ∗ f − e ∗ c;
  intersect = 0;
  if (det > 0) {
    if (((0.0 ≤ tmp1) ∧ (tmp1 ≤ det)) ∧ ((0.0 ≤ tmp2) ∧ (tmp2 ≤ det))) intersect = 1;
  }
  else if (det < 0) {
    if (((det ≤ tmp1) ∧ (tmp1 ≤ 0.0)) ∧ ((det ≤ tmp2) ∧ (tmp2 ≤ 0.0))) intersect = 1;
  }
  return (intersect);
}
```

This code is used in section 23.

40. Routine for determining whether a line and a box intersect or not. The *box_intersection*( ) routine simply divides the input box, being characterized by its lower-left and upper-right corners (llx, lly) and (urx, ury), into the four line segments corresponding to its four edges, followed by calling the routine *lines_intersect*( ) to determine if any of these edges intersect the line segment. If an intersection is found, the *box_intersection*( ) routine returns 1 (true), otherwise 0 (false).

Input variables:

px, *py*	The coordinates of the first end point $\mathbf{p} = (p_x, p_y)$ of the line segment.
qx, *qy*	The coordinates of the second end point $\mathbf{q} = (q_x, q_y)$ of the line segment.
llx, *lly*	Coordinates of the lower-left corner of the box.
urx, *ury*	Coordinates of the upper-right corner of the box.

Output variables:

On exit, the routine returns 1 if an intersection is found, otherwise 0 is returned, in either case the value are returned as integers of **short** precision.

⟨ Routine for determining whether a line and a box intersect or not 40 ⟩ ≡

```
short int box_intersection(double px, double py, double qx, double qy, double llx, double
        lly, double urx, double ury)
{
  if (lines_intersect(px, py, qx, qy, llx, lly, urx, lly)) return (1);      /* intersection with bottom edge */
  if (lines_intersect(px, py, qx, qy, urx, lly, urx, ury)) return (1);      /* intersection with right edge */
  if (lines_intersect(px, py, qx, qy, urx, ury, llx, ury)) return (1);      /* intersection with top edge */
  if (lines_intersect(px, py, qx, qy, llx, ury, llx, lly)) return (1);      /* intersection with left edge */
  return (0);      /* this happens only if no edge is intersecting the line segment */
}
```

This code is used in section 23.

41. Routines for calculation of number of boxes covering the trajectory. There are two almost identical routines for the calculation of the number of boxes covering the input trajectory at a given level of subdivision of the box sizes. The first routine, *get_num_covering_boxes*( ) simply performs this task without caring of documenting the box distributions as graphs, or "box maps", while the second one, *get_num_covering_boxes_with_boxmaps*( ) also includes the generation of these maps, with output in terms of METAPOST code.

⟨ Routines for calculation of number of boxes covering the trajectory 41 ⟩ ≡

```
⟨ Routine for calculation of number of boxes covering the trajectory 42 ⟩
⟨ Simplified routine for calculation of number of boxes covering the trajectory 52 ⟩
```

This code is used in section 23.

42. Routine for calculation of number of boxes covering the trajectory, also including the generation of documenting graphs of the box distributions. The *get_num_covering_boxes_with_boxmaps*( ) routine takes a trajectory as described by a discrete set of coordinates as input, and for a given grid refinement N returns the number of boxes needed to entirely cover the trajectory. The grid refinement factor N indicates that the domain of computation is divided into a $[2^N \times 2^N]$-grid of boxes.

The computational domain in which the box counting is to be performed is explicitly stated by the coordinates of its lower-left and upper-right corners, $(global_llx, global_lly)$ and $(global_urx, global_ury)$, respectively. Parts of the trajectory which are outside of this domain are not included in the box-counting. If the entire input trajectory is to be used in the box counting, simply use $(global\_llx, global\_lly) = (\min[x\_traj], \min[y\_traj])$ and $(global\_urx, global\_ury) = (\max[x\_traj], \max[y\_traj])$ for the specification of the computational domain.

Input variables:

mm	The total number of coordinates forming the input trajectory, or equivalently the length of the vectors $*x\_traj$ and $*y\_traj$.
$*x\_traj$, $*y\_traj$	Vectors of length *mm* containing the x- and y-coordinates of the input trajectory.
resolution	The grid refinement factor N.
global_llx, *global_lly*	Coordinates of the lower-left corner of the computational domain in which the box-counting is to be performed.
global_urx, *global_ury*	Coordinates of the upper-right corner of the computational domain in which the box-counting is to be performed.
trajectory_filename	String containing the name of the file containing the input trajectory.
boxgraph_filename	String which, if non-empty, states the file name to which the map of the spatial distribution of the covering boxes is to be written, as METAPOST code.

Output variables:

	On exit, the routine returns the number of covering boxes as an integer of **long unsigned** precision.

Internal variables:

px,*py*,*qx*,*qy*	Keeps track of the x- and y-coordinates of the start and end points of line segments, between $\mathbf{p} = (p_x, p_y)$ and $\mathbf{q} = (q_x, q_y)$.

```
⟨ Routine for calculation of number of boxes covering the trajectory 42 ⟩ ≡
  long unsigned int get_num_covering_boxes_with_boxmaps(double *x_traj, double *y_traj, long
        int mm, int resolution, double global_llx, double global_lly, double global_urx, double
        global_ury, char boxgraph_filename[], double boxgraph_width, double boxgraph_height, char
        trajectory_filename[])
  {
    short int **box;
    long unsigned int i, j, m, nn, istart, jstart, istop, jstop, iincr, jincr, num_boxes;
    double *x_box, *y_box;      /* Keeps track of the lower-left coordinates of the boxes. */
    double px, py, qx, qy;
    FILE *boxgraph_file;
    ⟨ Set up the grid of 2^N × 2^N boxes covering the entire global window of computation 43 ⟩
    ⟨ Find indices (i_a, j_a) of the box covering the first coordinate of the trajectory 44 ⟩
    for (m = 1; m ≤ mm − 1; m++) {
      ⟨ Find indices (i_b, j_b) of the box covering the end point of mth trajectory segment 45 ⟩
      ⟨ Scan the mth trajectory segment for intersecting boxes 46 ⟩
    }
    ⟨ Open file for output of box distribution graph 47 ⟩
    ⟨ Write the input trajectory to the box distribution graph 48 ⟩
    ⟨ Count the total number of boxes num_boxes intersecting the trajectory 49 ⟩
```

```
⟨ Write closing blocks the box distribution graph 50 ⟩
⟨ Close any open file for output of box distribution graph 51 ⟩
return (num_boxes);
}
```

This code is used in section 41.

43. Set up the grid of $2^N \times 2^N$ boxes covering the entire global window of computation. In this block, the resolution of the grid of boxes is defined. Notice that in many cases, only a certain subset of boxes will actally intersect the input trajectory. However, we do not á priori know this number of boxes, and in order to safeguard against the possible danger of running out of allocated memory, with the need for a dynamic memory allocation depending on the state of computation, a matrix of short integer precision is allocated covering the entire computational window. In order to keep track of the coordinates of the boxes, two vectors $x\_box[1 \ldots 2^N]$ and $y\_box[1 \ldots 2^N]$ are allocated to contain the x- and y-coordinates of the lower-left corners of the boxes, with the last elements $x\_box[2^N]$ and $y\_box[2^N]$ containing the upper-right corner coordinates of the upper-right-most box of the global window.

```
⟨ Set up the grid of 2^N × 2^N boxes covering the entire global window of computation 43 ⟩ ≡
{
  nn = 1;
  for (i = 1; i ≤ resolution; i++) nn = 2 * nn;
  box = simatrix(1, nn, 1, nn);
  for (i = 1; i ≤ nn; i++)
    for (j = 1; j ≤ nn; j++) box[i][j] = 0;
  x_box = dvector(1, nn + 1);
  y_box = dvector(1, nn + 1);
  for (m = 1; m ≤ nn + 1; m++) {
    x_box[m] = global_llx + ((double)(m − 1)) * (global_urx − global_llx)/((double)(nn));
    y_box[m] = global_lly + ((double)(m − 1)) * (global_ury − global_lly)/((double)(nn));
  }
}
```

This code is used in section 42.

44. Find indices (i_a, j_a) of the box covering the first coordinate of the trajectory.

```
⟨ Find indices (i_a, j_a) of the box covering the first coordinate of the trajectory 44 ⟩ ≡
{
  istart = 0;
  jstart = 0;
  for (m = 1; m ≤ nn; m++) {
    if ((x_box[m] ≤ x_traj[1]) ∧ (x_traj[1] ≤ x_box[m + 1])) istart = m;
    if ((y_box[m] ≤ y_traj[1]) ∧ (y_traj[1] ≤ y_box[m + 1])) jstart = m;
  }
  if ((istart ≡ 0) ∨ (jstart ≡ 0)) {
    fprintf(stderr, "%s: Error! Cannot find box indices of 1st coordinate!\n", progname);
    fprintf(stderr, "%s: Please check data for input trajetory.\n", progname);
    exit(FAILURE);
  }
}
```

This code is used in section 42.

45. Find indices (i_b, j_b) of the box covering the end point of the mth trajectory segment.

```
⟨ Find indices (i_b, j_b) of the box covering the end point of mth trajectory segment 45 ⟩ ≡
  {
    px = x_traj[m];
    py = y_traj[m];
    qx = x_traj[m + 1];
    qy = y_traj[m + 1];
    istop = istart;
    jstop = jstart;
    if (px < qx) {
      iincr = 1;
      while (x_box[istop + 1] < qx) istop++;
    }
    else {
      iincr = −1;
      while (x_box[istop] > qx) istop−−;
    }
    if (py < qy) {
      jincr = 1;
      while (y_box[jstop + 1] < qy) jstop++;
    }
    else {
      jincr = −1;
      while (y_box[jstop] > qy) jstop−−;
    }
    if (0 ≡ 1) {
      fprintf(stdout, "%s: Endpoint box indices: (i=%ld,j=%ld)\n", progname, istop, jstop);
    }
  }
```

This code is used in section 42.

46. Scan the mth trajectory segment for intersecting boxes. As the indices of the boxes covering the start and end points of the mth trajectory segment have been previously determined, one may now use this information in order to reduce the search for intersecting boxes to the domain as covered by box indices $i_{\text{start}} \le i \le i_{\text{stop}}$ and $j_{\text{start}} \le j \le j_{\text{stop}}$. This way, the computational time needed is greatly reduced as compared to if the entire window would be scanned for every segment.

```
⟨ Scan the mth trajectory segment for intersecting boxes 46 ⟩ ≡
  {
    for (i = istart; i ≠ (istop + iincr); i += iincr) {
      for (j = jstart; j ≠ (jstop + jincr); j += jincr) {
        if (box_intersection(px, py, qx, qy, x_box[i], y_box[j], x_box[i + 1], y_box[j + 1])) {
          box[i][j] = 1;
        }
      }
    }
    istart = istop;
    jstart = jstop;
  }
```

This code is used in section 42.

47. Open file for output of box distribution graph. In this block the preamble of the output METAPOST code is written to file. This preamble contains a macro `box(i,j)` which simply is a parameter-specific METAPOST subroutine which is used in order to reduce the final size of the code to be compiled into a graph over the distribution of covering boxes.

```
⟨ Open file for output of box distribution graph 47 ⟩ ≡
  {
    if (¬strcmp(boxgraph_filename, "")) {      /* is boxgraph_filename an empty string? */
      boxgraph_file = Λ;
    }
    else {
      if ((boxgraph_file = fopen(boxgraph_filename, "w")) ≡ Λ) {
        fprintf(stderr, "%s: Could not open %s for box graphs!\n", progname, boxgraph_filename);
        exit(FAILURE);
      }
      fseek(boxgraph_file, 0L, SEEK_SET);
      fprintf(boxgraph_file, "input graph;\n");
      fprintf(boxgraph_file, "def box(expr i,j)=\n");
      fprintf(boxgraph_file, "begingroup\n");
      fprintf(boxgraph_file, "llx:=%2.8f+(i-1)*%2.8f;\n", global_llx, (global_urx − global_llx)/(nn));
      fprintf(boxgraph_file, "lly:=%2.8f+(j-1)*%2.8f;\n", global_lly, (global_ury − global_lly)/(nn));
      fprintf(boxgraph_file, "urx:=%2.8f+(i)*%2.8f;\n", global_llx, (global_urx − global_llx)/(nn));
      fprintf(boxgraph_file, "ury:=%2.8f+(j)*%2.8f;\n", global_lly, (global_ury − global_lly)/(nn));
      fprintf(boxgraph_file, "gdraw (llx,lly)--(urx,lly);\n");
      fprintf(boxgraph_file, "gdraw (urx,lly)--(urx,ury);\n");
      fprintf(boxgraph_file, "gdraw (urx,ury)--(llx,ury);\n");
      fprintf(boxgraph_file, "gdraw (llx,ury)--(llx,lly);\n");
      fprintf(boxgraph_file, "endgroup\n");
      fprintf(boxgraph_file, "enddef;\n");
      fprintf(boxgraph_file, "beginfig(1);\n");
      fprintf(boxgraph_file, "w:=%2.4fmm; h:=%2.4fmm;\n", boxgraph_width, boxgraph_height);
      fprintf(boxgraph_file, "draw begingraph(w,h);\n");
      fprintf(boxgraph_file, "pickup pencircle scaled .3pt;\n");
      fprintf(boxgraph_file, "setrange(%2.6f,%2.6f,%2.6f,%2.6f);\n", global_llx, global_lly, global_urx,
          global_ury);
    }
  }
```

This code is used in section 42.

48. Write the input trajectory to the box distribution graph. Here there are two possible choices for how the input trajectory is to be included in the box graph.

First, we may use METAPOST to automatically scan and draw the trajectory for us, simply by using a statment like `gdraw input.dat`, assuming that the file `input.dat` contains properly formatted columnwise data. However, this choice would have two major drawbacks, namely that the generated code would be dependent on that the original input file always is in place, hence not allowing the METAPOST code to be exported as a standalone code as such, and also that this would put a limitation on the number of nodes allowed in the input trajectory, as the `graph.mp` macro package of METAPOST only accepts roughly 4000 points before it cuts the mapping.

The other choice is to include the input trajectory directly into the generated code, preferrably by chopping the trajectory into pieces small enough to easily be mapped by METAPOST without reaching a critical limit of exhaustion. This choice of course significantly increases the file size of the generated code, but this is a price we will have to accept in order to get stand-alone output. In the BOXCOUNT program, the second alternative was naturtally chosen, simply because the author is a fan of self-sustaining code which can be exported for later use.

In this block, the status of the pointer *boxgraph_file* is used to determine whether to write the trajectory to file or not. If *boxgraph_file* equals to Λ (NULL), then the BOXCOUNT program will not attempt to write the input trajectory to file.

```
⟨ Write the input trajectory to the box distribution graph 48 ⟩ ≡
  {
    if (boxgraph_file ≠ Λ) {
      fprintf(boxgraph_file, "pickup␣pencircle␣scaled␣.5pt;\n");
      i = 0;
      for (m = 1; m ≤ mm; m++) {
        if (i ≡ 0) {
          if (m ≡ 1) {
            fprintf(boxgraph_file, "gdraw␣(%2.4f,%2.4f)", x_traj[m], y_traj[m]);
          }
          else if (2 < mm) {
            fprintf(boxgraph_file, "gdraw␣(%2.4f,%2.4f)--(%2.4f,%2.4f)", x_traj[m − 1], y_traj[m − 1],
                x_traj[m], y_traj[m]);
          }
        }
        else {
          fprintf(boxgraph_file, "--(%2.4f,%2.4f)", x_traj[m], y_traj[m]);
        }
        i++;
        if ((i ≡ 5) ∨ (m ≡ mm)) {
          fprintf(boxgraph_file, ";\n");
          i = 0;
        }
      }
    }
  }
```

This code is used in section 42.

49. Count the total number of boxes *num_boxes* intersecting the trajectory. Having traversed the entire trajectory at the current depth of resolution, the only remaining task is to sum up the total number of boxes needed to cover the entire trajectory. This is simply done by extracting the number of set elements in the *box* matrix. Having extracted the total number of boxes, this block also takes care of releasing the memory occupied by the *box* matrix, as this memory might be needed for iterations to come in which an even finer mesh of boxes is used.

```
⟨ Count the total number of boxes num_boxes intersecting the trajectory 49 ⟩ ≡
  {
    num_boxes = 0;
    for (i = 1; i ≤ nn; i++) {
      for (j = 1; j ≤ nn; j++) {
        if (box[i][j] ≡ 1) {
          num_boxes++;
          if (boxgraph_file ≠ Λ) {
            fprintf(boxgraph_file, "box(%ld,%ld);\n", i, j);
          }
        }
      }
    }
    free_simatrix(box, 1, nn, 1, nn);
  }
```

This code is used in section 42.

50. Write closing blocks the box distribution graph. Here follows the specification of tick marking (set as automatic for the sake of simplicity) and axis labels, just before the closing statements of the METAPOST code for the graphs of box distributions.

```
⟨ Write closing blocks the box distribution graph 50 ⟩ ≡
  {
    if (boxgraph_file ≠ Λ) {
      fprintf(boxgraph_file, "autogrid(itick␣bot,itick␣lft);\n");
      fprintf(boxgraph_file, "glabel.bot(btex␣$x$␣etex,OUT);\n");
      fprintf(boxgraph_file, "glabel.lft(btex␣$y$␣etex,OUT);\n");
      fprintf(boxgraph_file, "endgraph;\n");
      fprintf(boxgraph_file, "endfig;\n");
      fprintf(boxgraph_file, "end\n");
    }
  }
```

This code is used in section 42.

51. Close any open file for output of box distribution graph.

```
⟨ Close any open file for output of box distribution graph 51 ⟩ ≡
  {
    if (boxgraph_file ≠ Λ) {
      fprintf(stdout, "%s:␣MetaPost␣output␣box␣distribution␣graph␣written␣to␣%s.\n", progname,
          boxgraph_filename);
      fclose(boxgraph_file);
    }
  }
```

This code is used in section 42.

52. Routine for calculation of number of boxes covering the trajectory. This routine provides a simplified interface to the general boxcountng routine *get_num_covering_boxes_with_boxmaps*, with the only difference that no graph of the box distribution over the trajectory is generated. The *get_num_covering_boxes*( ) routine takes a trajectory as described by a discrete set of coordinates as input, and for a given grid refinement N returns the number of boxes needed to entirely cover the trajectory. The grid refinement factor N indicates that the domain of computation is divided into a $[2^N \times 2^N]$-grid of boxes.

The computational domain in which the box counting is to be performed is explicitly stated by the coordinates of its lower-left and upper-right corners, $(global_llx, global_lly)$ and $(global_urx, global_ury)$, respectively. Parts of the trajectory which are outside of this domain are not included in the box-counting. If the entire input trajectory is to be used in the box counting, simply use $(global\_llx, global\_lly) = (\min[x\_traj], \min[y\_traj])$ and $(global\_urx, global\_ury) = (\max[x\_traj], \max[y\_traj])$ for the specification of the computational domain.

Input variables:

mm	The total number of coordinates forming the input trajectory, or equivalently the length of the vectors $*x\_traj$ and $*y\_traj$.
$*x\_traj$, $*y\_traj$	Vectors of length *mm* containing the x- and y-coordinates of the input trajectory.
resolution	The grid refinement factor N.
global_llx, *global_lly*	Coordinates of the lower-left corner of the computational domain in which the box-counting is to be performed.
global_urx, *global_ury*	Coordinates of the upper-right corner of the computational domain in which the box-counting is to be performed.

Output variables:

	On exit, the routine returns the number of covering boxes as an integer of **long unsigned** precision.

```
⟨ Simplified routine for calculation of number of boxes covering the trajectory 52 ⟩ ≡
  long unsigned int get_num_covering_boxes(double *x_traj, double *y_traj, long int mm, int i, double
        global_llx, double global_lly, double global_urx, double global_ury)
  {
    return (get_num_covering_boxes_with_boxmaps(x_traj, y_traj, mm, i, global_llx, global_lly, global_urx,
        global_ury, "", 0.0, 0.0, ""));
  }
```

This code is used in section 41.

53. Index.

⟨ Close any open file for output of box distribution graph 51 ⟩ Used in section 42.
⟨ Close file for output of logarithmic estimate 21 ⟩ Used in section 7.
⟨ Compute the logarithmic estimate of the fractal dimension 19 ⟩ Used in section 7.
⟨ Count the total number of boxes *num_boxes* intersecting the trajectory 49 ⟩ Used in section 42.
⟨ Declaration of local variables 11 ⟩ Used in section 7.
⟨ Display elapsed execution time 22 ⟩ Used in section 7.
⟨ Display starting time of program execution 14 ⟩ Used in section 7.
⟨ Extract boundary of global window of computation from input trajectory 17 ⟩ Used in section 7.
⟨ Find indices (i_a, j_a) of the box covering the first coordinate of the trajectory 44 ⟩ Used in section 42.
⟨ Find indices (i_b, j_b) of the box covering the end point of mth trajectory segment 45 ⟩ Used in section 42.
⟨ Get number of boxes covering the trajectory for all levels of refinement in resolution 18 ⟩ Used in section 7.
⟨ Global definitions 9 ⟩ Used in section 7.
⟨ Global variables 10 ⟩ Used in section 7.
⟨ Initialize variables 12 ⟩ Used in section 7.
⟨ Library inclusions 8 ⟩ Used in section 7.
⟨ Load input trajectory from file 15 ⟩ Used in section 7.
⟨ Open file for output of box distribution graph 47 ⟩ Used in section 42.
⟨ Open file for output of logarithmic estimate 16 ⟩ Used in section 7.
⟨ Parse command line for parameters 13 ⟩ Used in section 7.
⟨ Routine for allocation of matrices of short integer precision 36 ⟩ Used in section 29.
⟨ Routine for allocation of vectors of double precision 30 ⟩ Used in section 29.
⟨ Routine for allocation of vectors of integer precision 32, 34 ⟩ Used in section 29.
⟨ Routine for calculation of number of boxes covering the trajectory 42 ⟩ Used in section 41.
⟨ Routine for checking valid path characters 27 ⟩ Used in section 26.
⟨ Routine for computation of average, average deviation and standard deviation 24 ⟩ Used in section 23.
⟨ Routine for deallocation of matrices of short integer precision 37 ⟩ Used in section 29.
⟨ Routine for deallocation of vectors of double precision 31 ⟩ Used in section 29.
⟨ Routine for deallocation of vectors of integer precision 33, 35 ⟩ Used in section 29.
⟨ Routine for determining whether a line and a box intersect or not 40 ⟩ Used in section 23.
⟨ Routine for determining whether two lines intersect or not 39 ⟩ Used in section 23.
⟨ Routine for displaying help message 38 ⟩ Used in section 23.
⟨ Routine for obtaining the number of coordinate pairs in a file 25 ⟩ Used in section 23.
⟨ Routine for stripping away path string 28 ⟩ Used in section 26.
⟨ Routines for calculation of number of boxes covering the trajectory 41 ⟩ Used in section 23.
⟨ Routines for memory allocation of vectors and matrices 29 ⟩ Used in section 23.
⟨ Routines for removing preceding path of filenames 26 ⟩ Used in section 23.
⟨ Save or append the logarithmic estimate to output file 20 ⟩ Used in section 7.
⟨ Scan the mth trajectory segment for intersecting boxes 46 ⟩ Used in section 42.
⟨ Set up the grid of $2^N \times 2^N$ boxes covering the entire global window of computation 43 ⟩ Used in section 42.
⟨ Simplified routine for calculation of number of boxes covering the trajectory 52 ⟩ Used in section 41.
⟨ Subroutines 23 ⟩ Used in section 7.
⟨ Write closing blocks the box distribution graph 50 ⟩ Used in section 42.
⟨ Write the input trajectory to the box distribution graph 48 ⟩ Used in section 42.

BOXCOUNT